Title:	Review of “SClib, a hack for straightforward embedded C functions in (I)Python”
Author(s):	Certik, Ondrej
Intended for:	Public review of a journal paper
Issued:	2014-08-28

Review of “SClib, a hack for straightforward embedded C functions in (I)Python”

Paper available at: https://github.com/euroscipy/euroscipy_proceedings/pull/39
Commit that was reviewed: b7257d4dd240d9cd008260c68f13a7e47f79e3a2

The authors present a simple library SClib for creating Python wrappers to C functions. From the user perspective, one uses C macros to declare how things should be wrapped on the C level, and then SClib takes care of the rest. The authors then provide two applications of SClib: radial Schrödinger eigensolver and control engineering application.

The second control engineering application does not have enough numerical details in order to reproduce the graphs, but the goal of that section is to provide users with a high level overview of how SClib can be used in control engineering (to efficiently evaluate derivatives and other functions) --- for that purpose the level of detail seems sufficient.

The first application in quarkonium physics contains enough detail in order to reproduce most results. Using the program dftatom described in [1], I have verified that the energies in Table I are correct to all printed digits. I have created an IPython Notebook, available at [2], that uses the Python interface to dftatom and I am providing it as part of this review, so that it is clear exactly how the results were obtained.

Citing from the abstract, the goal of this first application is to “efficiently solve the Schrödinger equation for bounds states” and drive the calculation in IPython (Notebook) so that one can easily manipulate the solutions and visualize results in an interactive environment, without sacrificing speed. Table I shows time comparisons of various radial Schrödinger solvers.

In order to asses the claim whether the speed was sacrificed or not, the authors should compare against other codes that people have used in literature. One could start for example with the list of solvers provided in Table 1 in [1]. In order to get a sense of this, I calculated the four states in Table I using dftatom [1], and with optimized mesh for this problem it seems to be more than 10,000 times faster, i.e. four orders of magnitude. The calculation, plots, and timings are provided in the Notebook [2].

How is the radial mesh determined in the manuscript? I.e. do the timings in Table I include a single fixed mesh calculation (just like it is provided for `dftatom` in [2]), or does it include adaptive mesh refinement until the solution is converged? In order to make fair comparison with `dftatom`, one needs to make sure to either: (1) both use the same fixed mesh, or (2) both determine the mesh adaptively.

Even though dftatom in [1] is using a shooting method and bisection, just like the authors used in this paper, one could argue that it is implemented in a more efficient way than the Mathematica code in [3] or the `SChroe.py` script in this paper. However, given that it is the same method, four orders of magnitude is such a big speedup that the claim that `SChroe.py` solves the Schrödinger equation efficiently is in my opinion not warranted. The authors should mention this fact, that by implementing the shooting method efficiently in Fortran or C and using a fixed mesh calculation, one can obtain four orders of magnitude speedup. Or they can just add the dftatom’s timings (or of another similarly fast solver) as another column into Table I, assuming that those timings are for a single fixed mesh calculation.

Error in equation: Equation (1) presents Schrödinger equation with reduced mass `m`, and the term 1/2. Then the radial equation in (2) must also have the term 1/2. As it is now, the `m` in equation (2) is twice the `m` in equation (1). As such, the authors should rename the `m` in (1) e.g. to `mu` and clarify that `mu=m/2`. It is more conventional to use `mu`, not `m`. Since the authors use `m` in their paper, when reproducing their results, it is essential to use `mu=m/2` in order to agree.

The graph of the wavefunction in Fig.1 b appears to be incorrectly normalized. Using the usual normalization, the correct graph should look similar to the one in the cell [9] in [1], i.e. notice the different magnitude/normalization.

The paper should also mention how SClib compares against other tools for creating Python wrappers like Cython or Swig.

I noted the following typos/grammatical errors:

- “The functions are then available as a members of the library” -> “The functions are then available as members of the library” (no “a”)
- in abstract: “conceading” -> “conceding”, but the sentence doesn’t make sense, perhaps the authors meant “compromise”?
- The code for SClib and example use are availible at -> available
- “once the paper appear online” -> “once the paper appears online”
- “With the goal of mimic the advantages of this script but without compromising in speed we have developed SChroe.py,” -> “With the goal of mimicking the advantages of this script, but without compromising speed, we have developed SChroe.py” (i.e. two commas missing, “mimicking” and no “in”)
- “The Schrödinger equation is the one the fundamental equations in physics” perhaps the authors meant “one of the fundamental equations” ?
- “we will focus on the time-independent version which in natural units is given by“ -> “we will focus on the time-independent version which, in natural units, is given by”
- “If the relative velocity of between the quark and the antiquark” perhaps remove “between”?
- “the value of :math:`E_{n,l}` correspond to an eigenvalue” -> corresponds
- “fulfills the condition of have one node” -> “fulfills the condition of having one node”
- “the standard method consist in to apply” perhaps “the standard method consist of applying” sounds better
- “In general :math:`y_{n,l}(r)` will diverge except when :math:`E_{n,l}` correspond to an eigenvalue.” -> corresponds

And several formatting issues:

- `Schr\"odinger` doesn’t render correctly, the authors have to write it as Schrödinger
- Fig.2: `ref{cornell}` doesn’t render correctly. The same on the page 50.
- `ref{coupled}`, page 52 doesn’t render correctly
- section 3 has `latex-preamble: DeclareMathOperator*{argmin}{arg,min}` in the pdf.

After addressing the above issues, I think this paper can be accepted.

Ondřej Čertík
CCS-2: Computational Physics and Methods
Los Alamos National Laboratory
LA-UR: ###

[1] Čertík, O., Pask, J. E., & Vackář, J. (2013). dftatom: A robust and general Schrödinger and Dirac solver for atomic structure calculations. Computer Physics Communications, 184(7), 1777–1791. doi:10.1016/j.cpc.2013.02.014

[2] ### This will be a link to the notebook at http://nbviewer.ipython.org/ ### For now the notebook is attached as pdf.

[3] Lucha, W., & Schöberl, F. F. (1999). Solving the Schrödinger Equation for Bound States With Mathematica 3.0. International Journal of Modern Physics C, 10(04), 607–619. doi:10.1142/S0129183199000450

To get a sense for relative speed and ease of use, let's reproduce Ref. [1] Table 1 using the available package dftatom [2]."

Note: `dftatom` is working in Hartree atomic units, so we first need to convert the radial Schödinger equation with reduced mass μ (other than 1 in a.u.) into atomic units. This can be done by multiplying the potential by μ and then dividing the calculated energy by μ.

Also note that the paper https://github.com/euroscipy/euroscipy_proceedings/pull/39 uses n' (I use prime to distinguish it from n used in this notebook) which always starts at 0 and counts the states for the given l, as is the convention e.g. for harmonic oscillator. In atomic physics (e.g. in dftatom), on the other hand, the convention is to start n at 1 and l is from $0 \le l < n$. The relation is $n' = n - l - 1$.

[1] Lucha, W., Schöberl, F. F. (1999). Solving the Schrödinger Equation for Bound States With Mathematica 3.0. International Journal of Modern Physics C, 10(04), 607–619. doi:10.1142/S0129183199000450

[2] Čertík, O., Pask, J. E., Vackář, J. (2013). dftatom: A robust and general Schrödinger and Dirac solver for atomic structure calculations. Computer Physics Communications, 184(7), 1777–1791. doi:10.1016/j.cpc.2013.02.014.

In [1]:
```
%matplotlib inline
from math import sqrt
from timeit import default_timer as clock
import matplotlib.pyplot as plt
from numpy import empty
from dftatom import (mesh_exp, mesh_exp_deriv, solve_radial_eigenproblem,
atom_lda, atom_rlda)
```

```
In [2]: a = 2.7e6
        rmin = 1e-7
        rmax = 50
        N = 5000
        relat = 0
        R = mesh_exp(rmin, rmax, a, N)
        Rp = mesh_exp_deriv(rmin, rmax, a, N)
        V_tot = R**2
        mu = 0.5 # reduced mass
        print "n-l-1  l      E"
        for n in range(5):
            for l in range(n):
                E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                        V_tot*mu, R, Rp, 0, relat, False, 0, 100)
                print "  %d    %d %10.6f" % (n-l-1, l, E/mu)
```

```
n-l-1  l      E
  0    0   3.000000
  1    0   7.000000
  0    1   5.000000
  2    0  11.000000
  1    1   9.000000
  0    2   7.000000
  3    0  15.000000

  2    1  13.000000
  1    2  11.000000
  0    3   9.000000
```

This agrees exactly with Table 1. Let's reproduce Table 2 in [1]:

```
In [3]: Z = 1
        V_tot = -Z/R
        mu = 0.5
        print "n  l     -E/2"
        for n in range(3):
            for l in range(n):
                E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                        V_tot*mu, R, Rp, Z, relat, True, -100, 0)
                print "%d  %d %10.6f" % (n, l, -E/mu)
```

```
n  l     -E/2
1  0   0.250000
2  0   0.062500
2  1   0.062500
```

Again, this agrees exactly. Finally, let's reproduce the Table 1 in https://github.com/euroscipy/euroscipy_proceedings/pull/39:

```
In [4]: m = 1.
        a = 0.1
        k = 0.5*m**2

        V_tot = a/R + k*R
        mu = m/2 # reduced mass
        print "n-l-1  l      E    time[s]"
        for n in range(25):
            for l in range(n):
                if l != 1: continue
                if n-l-1 not in [0, 1, 2, 20]: continue
                t1 = clock()
                E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                        V_tot*mu, R, Rp, 0, relat, False, 0, 100)
                t2 = clock()
                print "  %2d   %d %10.5f  %.2f" % (n-l-1, l, E / mu, t2-t1)
```

```
n-l-1  l      E    time[s]
   0   1    2.15789  0.01
   1   1    3.10952  0.01
   2   1    3.93850  0.01
  20   1   13.59946  0.01
```

The energies agree to all printed digits.

Let's get more accurate timing for the `n-l-1=20` case:

```
In [5]: %%timeit
        l = 1
        n = 20+l+1
        E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                        V_tot*mu, R, Rp, 0, relat, False, 0, 100)
```

```
100 loops, best of 3: 9.56 ms per loop
```

```
In [6]: E/mu
```

```
Out[6]: 13.599463957910984
```

Let's reproduce the Fig. 1:

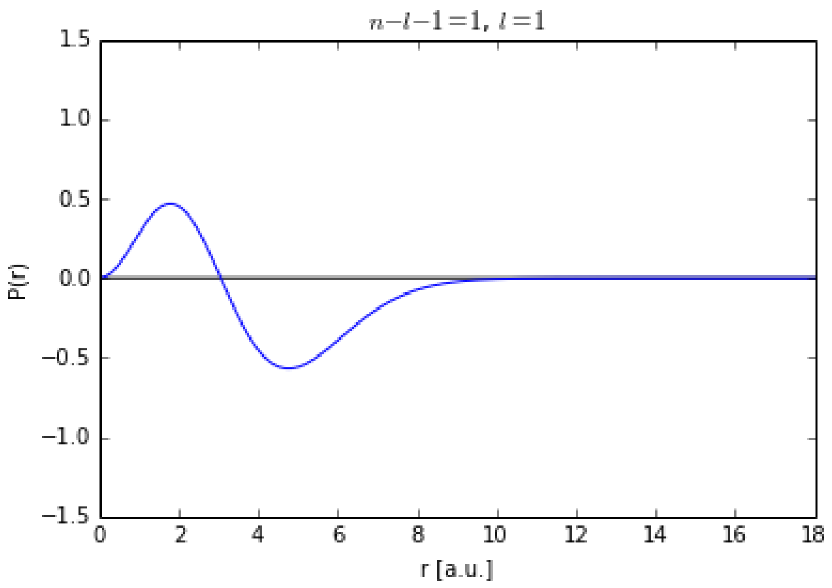
```
In [7]: l = 1
        n = 1+l+1
        E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                        V_tot*mu, R, Rp, 0, relat, False, 0, 100)
        E/mu
```

```
Out[7]: 3.109521596201148
```

```
In [8]: E/mu
```

```
Out[8]: 3.109521596201148
```

```
In [9]:  plt.plot(R, [0]*len(R), "k-")
         plt.plot(R, P, "b-")
         plt.xlim([0, 18])
         plt.ylim([-1.5, 1.5])
         plt.xlabel("r [a.u.]")
         plt.ylabel("P(r)")
         plt.title("$n-l-1=1$, $l=1$")
         plt.show()
```

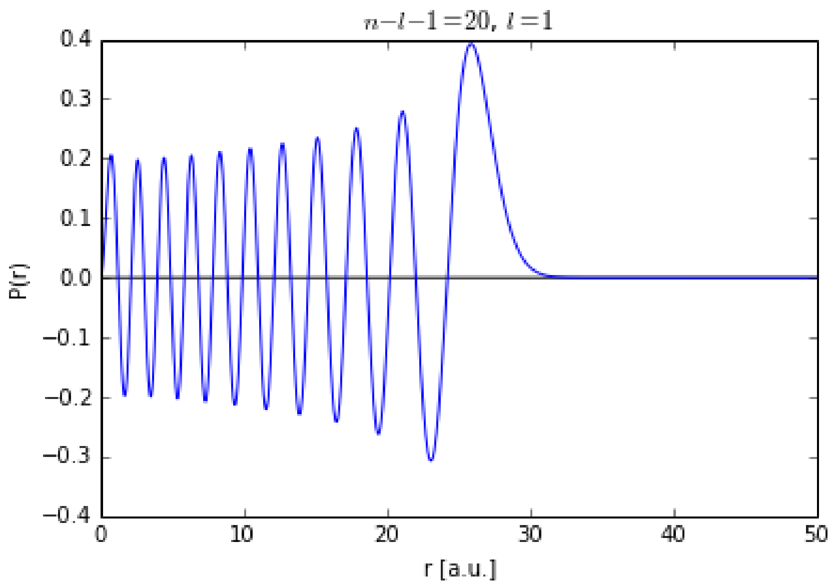


The difference in magnitude is due to the fact that `dftatom` returns normalized wavefunctions. Let's also plot the `n-l-1=20` wavefunction:

```
In [10]:  l = 1
          n = 20+l+1
          E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                          V_tot*mu, R, Rp, 0, relat, False, 0, 100)
          E/mu

Out[10]:  13.599463957910984
```

In [11]:
```
plt.plot(R, [0]*len(R), "k-")
plt.plot(R, P, "b-")
plt.xlabel("r [a.u.]")
plt.ylabel("P(r)")
plt.title("$n-l-1=20$, $l=1$")
plt.show()
```



Optimal Mesh

One can obtain a more efficient mesh following the procedure in [xxx].

One obtains:

```
a=5
N=1500
rmin=1e-2
rmax=50
```

Let's rerun the benchmark with this mesh:

In [17]:
```
a = 5
rmin = 1e-2
rmax = 50
N = 1500
R = mesh_exp(rmin, rmax, a, N)
Rp = mesh_exp_deriv(rmin, rmax, a, N)
m = 1.
a = 0.1
k = 0.5*m**2
V_tot = a/R + k*R
mu = m/2 # reduced mass
print "n-l-1  l      E    time[s]"
for n in range(25):
    for l in range(n):
        if l != 1: continue

        if n-l-1 not in [0, 1, 2, 20]: continue
        t1 = clock()
        E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                V_tot*mu, R, Rp, 0, relat, False, 0, 100)
        t2 = clock()
        print "  %2d   %d %10.5f  %.2f" % (n-l-1, l, E / mu, t2-t1)
```

```
n-l-1  l      E    time[s]
   0   1    2.15789  0.00
   1   1    3.10952  0.00
   2   1    3.93850  0.00
  20   1   13.59948  0.00
```

In [18]:
```
%%timeit
l = 1
n = 20+l+1
E, P, Q = solve_radial_eigenproblem(0.0, n, l, -1, 1e-11, 100,
                V_tot*mu, R, Rp, 0, relat, False, 0, 100)
```

```
100 loops, best of 3: 2.81 ms per loop
```

In [19]:
```
E/mu
```

Out[19]:
```
13.599475459369614
```

So the accuracy didn't change, but as we can see for the `n-l-1=20` case, the calculation got from 9.4ms to 2.8ms, which is over 3x faster.

Compared to the `Schroe.py`, `dftatom` is now `32.13*1000/2.8=11475` more than 10,000 times faster.